Lazuli Documentation

Release 0.1.0

Remi Andruccioli

Jan 05, 2021

Contents

You are reading the documentation of the Lazuli project.

The project is hosted at https://github.com/randruc/Lazuli

This documentation is divided in two parts: **User** and **Kernel**.

The **User** documentation is intended for end users, who wish to develop real-time systems and applications using the Lazuli kernel. This documentation will guide you through the process of getting Lazuli sources, setting up a working development environment, and compiling your code with Lazuli. It will also give you detailed information about the functionalities provided by Lazuli.

The last chapter of this documentation is the **Kernel** documentation. It is intended for programmers, who wish to code in the Lazuli kernel, hacking and improving it. This documentation will give you detailed information about the inner working of Lazuli.

CHAPTER 1

Introduction

What is Lazuli, and what it is not.

Lazuli is a preemptive real-time multitasking kernel targeting microcontrollers, or machines with constrained resources. It allow the microcontroller to run multiple independent tasks simultaneously, with some of them having hard real-time constraints. Lazuli provides a "time slice" scheduler in order to respect deadlines constraints of the tasks running in the system.

For now only the ATmega328p is supported.

Lazuli is self-contained: it does not rely on an existing library.

Lazuli currently provides the following functionalities :

- **"ROMable"**: i.e. All the system can fit in ROM. Lazuli does not rely on the presence of a disk or storage device.
- **Real-time scheduling**: Tasks can be scheduled in a cyclic real-time Rate Monotonic Scheduling (RMS) fashion, or in a real-time priority round robin fashion (equivalent of POSIX SCHED_RR).
- **No MMU**: Lazuli does not relies on MMU or virtual memory. It runs on a unique flat address space, traditionally found in microcontrollers.
- **Modular**: Lazuli build system lets you choose which parts of the system you need, including only those necessary parts in the final binary, thus saving storage.
- **Containerized development environment**: An official container image is provided, that includes all the necessary tools to build the system and your own application.

CHAPTER 2

A simple example

Instead of the traditional “Hello world”, we show here a simple clock application.

This application demonstrates a simple clock that prints the time every second on the serial port.

```
/*
 * SPDX-License-Identifier: GPL-3.0-only
 * This file is part of Lazuli.
 */

/**
 * @file
 * @brief A simple clock.
 * @copyright 2019-2020, Remi Andruccioli <remi.andruccioli@gmail.com>
 *
 * An example program demonstrating a simple real-time task: a clock.
 */

#include <stdio.h>

#include <Lazuli/clock_24.h>
#include <Lazuli/common.h>
#include <Lazuli/lazuli.h>
#include <Lazuli/serial.h>

DEPENDENCY_ON_MODULE(CLOCK_24);
DEPENDENCY_ON_MODULE(PRINTF);
DEPENDENCY_ON_MODULE(SERIAL);

/**
 * Main clock task.
 */
void
ClockTask(void)
{
  Clock24 clock24;
```

(continues on next page)

(continued from previous page)

```
  for (;;) {
    Lz_Task_WaitActivation();

    Lz_Clock24_Get(&clock24);

    printf("%02u:%02u:%02u" LZ_CONFIG_SERIAL_NEWLINE,
           clock24.hours,
           clock24.minutes,
           clock24.seconds);
  }
}

/**
 * Main entry point for user tasks.
 */
void
main(void)
{
  Lz_TaskConfiguration taskConfiguration;
  Lz_SerialConfiguration serialConfiguration;

  /*
   * Enable serial transmission.
   */
  Lz_Serial_GetConfiguration(&serialConfiguration);
  serialConfiguration.enableFlags = LZ_SERIAL_ENABLE_TRANSMIT;
  serialConfiguration.speed = LZ_SERIAL_SPEED_19200;
  Lz_Serial_SetConfiguration(&serialConfiguration);

  /*
   * Configure scheduling parameters of the clock task.
   */
  Lz_TaskConfiguration_Init(&taskConfiguration);
  taskConfiguration.schedulingPolicy = CYCLIC_RT;
  taskConfiguration.period = 50;
  taskConfiguration.completion = 25;
  Lz_RegisterTask(ClockTask, &taskConfiguration);

  Lz_Run();
}
```

In this code sample, the `main` function is executed at system startup. In this function we configure the serial port to enable serial transmission and configure the speed of the transmission. Then we configure a single task to run. The entry point for this task is the function `ClockTask`. We configure this task to be real-time cyclic, with a period of 50 time units (time slices), and a completion time of 25 time units (time slices).

CHAPTER 3

Getting Lazuli

To start developing with Lazuli, the first thing to do is to get your own copy of the Lazuli source code.

The project is hosted on GitHub at https://github.com/randruc/Lazuli

3.1 Using Git

To get a copy of the Lazuli repository using Git, simply clone the GitHub repository:

```
git clone `<https://github.com/randruc/Lazuli.git>`_
```

3.2 Without Git

If you're not using Git, you can download a copy of the repository as a ZIP file here : https://github.com/randruc/Lazuli/archive/master.zip

You can then unzip it in any directory you wish.

CHAPTER 4

Set up the development environment

In order to develop with Lazuli you need to set up a development environment. For that it is strongly recommended to use the official Lazuli Docker image.

You can choose to install your own tools on your machine, however it is not guaranteed that the code will compile correctly. The Docker image integrates the exact versions needed for Lazuli to compile correctly. There are many advantages of using the official Lazuli Docker image:

1. The development of the Lazuli kernel itself is made using the official Lazuli Docker image. So no guarantee is given for the code to work as expected when compiled with versions of the tools that are different than the versions provided in the Lazuli Docker image.
2. As the Docker image is updated with the rest of the source code in the repository, you have the guarantee that the development environment always uses the right versions of the tools. Your development environment will keep updated with the successive versions of the repository.
3. You do not have anything to install on your development machine.
4. Using the Docker image allows you to develop with Lazuli whether your machine runs Linux, Windows or macOS.

In the rest of this documentation, we consider that you are using the official Lazuli Docker image.

4.1 About the Lazuli Docker image

You need to keep in mind a few things about the Lazuli Docker image.

The Docker image is used *interactively*. When the container is run, it starts a bash session that keeps alive for your whole development session. The user is root inside the container.

The Docker image uses a Fedora base image. This is why the Lazuli Docker image can be quite big. The image *does* integrate all the tools needed to configure build settings, to compile the kernel and your application, and to analyze the produced binaries. The Docker image *does not* integrate any text editor and no tool to upload the produced binary to the target machine. It does not integrate Git neither. Code editing and Git operations are done with your own tools, installed on your own host machine.

Finally, the Docker image does not integrate any copy of the repository or your own source code. Instead, your own copy of the repository will be mapped inside the running container. The file system of the Lazuli container is not persistent.

4.2 Getting the Lazuli Docker image

Lazuli Docker images are hosted in DockerHub at https://hub.docker.com/r/randruc/lazuli

4.2.1 Pulling the official image

You can use any tool that is compatible with OCI containers, like Docker or podman:

```
docker pull randruc/lazuli:latest
```

4.2.2 Building your own image

4.3 Run the container

Run the container with the following options. Remember to replace `/path/to/your/Lazuli/directory` with the actual path to your copy of the repository:

```
docker run --name lazuli -ti --rm -v/path/to/your/Lazuli/directory:/~/workspace:z␣
↪randruc/lazuli
```

You should see a message, followed by a bash prompt:

```
$ docker run --name lazuli -ti --rm -v$(pwd):/~/workspace:z randruc/lazuli
Welcome in the Lazuli development environment container.
For Lazuli version 0.1.0.
This image was generated on Mon May 11 20:09:37 UTC 2020.
Fedora release 31 (Thirty One)
Linux e654b59f6c18 5.6.13-100.fc30.x86_64 #1 SMP Fri May 15 00:36:06 UTC 2020 x86_64␣
↪x86_64 x86_64 GNU/Linux

[root@e654b59f6c18 workspace]#
```

The second line of the message indicates the corresponding version of Lazuli the image is built for. It should match the version number in the file named VERSION at the root of your copy of the Lazuli repository. If it doesn't, you are not using the appropriate Docker image, and you should pull the appropriate tag.

The fourth line indicates the Fedora version of the image.

The fifth line indicates the host system you are actually using.

CHAPTER 5

Developing your project

5.1 Your application code

At the root of the repository is a directory named `user`. This is where your own source code will go. A default file named `main.c` is provided. If you need to add more source files to your project, you must declare them in the file `CMakeLists.txt` at the root of the repository: you will add your files to the list named `LAZULI_USER_SOURCE_FILES`.

5.2 Building

The next step is to build your project.

The Lazuli kernel itself is not executable. It is developed as a static library that your own program will link against at compile time. This is why the `main` function that you can see in the file `main.c` is the *real* main function of the final program.

The whole Lazuli project relies on CMake for configuration and compilation. As Lazuli is designed to be cross platform and configurable, we will explain here how to configure CMake to your specific project before building it.

5.2.1 Configuration

The first step is to make CMake generate the appropriate cross compilation toolchain for your project. This toolchain depends on the target platform.

We will do that in an interactive fashion using the tool `ccmake`. It is recommended that you perform these operations in the Lazuli container.

We first create and browse the directory `build` which is the destination directory of build artefacts, and we invoke `cmake` by pointing to the root `CMakeFile.txt`

```
[root@6bf01305461f workspace]# mkdir build && cd build
[root@6bf01305461f build]# cmake ..
CMake Error at sys/cmake/machine_choice.cmake:23 (message):
Configuration error: The target machine must be defined with cache variable
'LZ_TARGET_MACHINE_CHOICE'.
Call Stack (most recent call first):
sys/CMakeLists.txt:23 (include)


-- Configuring incomplete, errors occurred!
[root@6bf01305461f build]#
```

This results in an error because the build system doesn't kow yet what is the target platform. We will fix this using `ccmake`, a console tool used to configure cmake cache:

```
[root@6bf01305461f build]# ccmake .
```

ccmake only displays one configuration variable, LZ_TARGET_MACHINE_CHOICE, by hitting ENTER (RETURN) will cycle through the supported platforms.

```
                                                  Page 1 of 1
LZ_TARGET_MACHINE_CHOICE









LZ_TARGET_MACHINE_CHOICE: Choice of the target machine.
Keys: [enter] Edit an entry [d] Delete an entry             CMake Version 3.17.2
    [l] Show log output   [c] Configure
    [h] Help              [q] Quit without generating
    [t] Toggle advanced mode (currently off)
```

For now, only AVR_ATmega328p is supported.

```
                                                  Page 1 of 1
LZ_TARGET_MACHINE_CHOICE         AVR_ATmega328p
```

(continues on next page)

(continued from previous page)

```
LZ_TARGET_MACHINE_CHOICE: Choice of the target machine.
Keys: [enter] Edit an entry [d] Delete an entry             CMake Version 3.17.2
    [l] Show log output   [c] Configure
    [h] Help              [q] Quit without generating
    [t] Toggle advanced mode (currently off)
```

We can now ask CMake to configure the cross-compilation toolchain, hitting `c`:

```
The C compiler identification is GNU 9.2.0
The ASM compiler identification is GNU
Found assembler: /usr/bin/avr-gcc
Check for working C compiler: /usr/bin/avr-gcc
Check for working C compiler: /usr/bin/avr-gcc - works
Detecting C compiler ABI info
Detecting C compiler ABI info - done
Detecting C compile features
Detecting C compile features - done
Using module: module_version_string
Configuring done




Configure produced the following output
                                                            CMake Version 3.17.2
Press [e] to exit screen
```

After exiting this screen by hitting `e`, ccmake now displays all the available configuration variables for the toolchain it created:

```
                                                Page 1 of 3
AVR_AR                          */usr/bin/avr-ar
AVR_CC                          */usr/bin/avr-gcc
AVR_NM                          */usr/bin/avr-nm
AVR_OBJCOPY                     */usr/bin/avr-objcopy
AVR_OBJDUMP                     */usr/bin/avr-objdump
CMAKE_BUILD_TYPE                *
CMAKE_INSTALL_PREFIX            */usr/local
CMAKE_TOOLCHAIN_FILE            */~/workspace/sys/cmake/avr.toolchain.cmake
LZ_CONFIG_AVR_INSTRUMENT_PORT   *0x2B
LZ_CONFIG_BREAK_STACK_GAP       *50
LZ_CONFIG_CHECK_INTERRUPT_CODE  *ON
LZ_CONFIG_CHECK_NULL_PARAMETER  *ON
```

(continues on next page)

(continued from previous page)

```
LZ_CONFIG_CHECK_NULL_PARAMETER  *ON
LZ_CONFIG_CHECK_NULL_PARAMETER  *ON
LZ_CONFIG_CHECK_NULL_PARAMETER  *ON
LZ_CONFIG_CHECK_WRONG_ENUM_ENT  *ON
LZ_CONFIG_DEFAULT_TASK_PRIORIT  *0

AVR_AR: Path to a program.
Keys: [enter] Edit an entry [d] Delete an entry             CMake Version 3.17.2
    [l] Show log output   [c] Configure
    [h] Help              [q] Quit without generating
    [t] Toggle advanced mode (currently off)
```

You can configure these variables according to your needs and hit `c` again:

```
Using module: module_version_string
Configuring done









Configure produced the following output
                                                            CMake Version 3.17.2
Press [e] to exit screen
```

After hitting `e`, you can now generate the full configured toolchain and exit by hitting `g`. You can then return to the workspace:

```
[root@6bf01305461f build]# cd ..
[root@6bf01305461f workspace]#
```

Note that if you already know your target platform, the cross compilation toolchain can be generated by doing only:

```
[root@6bf01305461f workspace]# cmake -S . -B ./build -DLZ_TARGET_MACHINE_CHOICE=AVR_
↪ATmega328p
-- The C compiler identification is GNU 9.2.0
-- The ASM compiler identification is GNU
-- Found assembler: /usr/bin/avr-gcc
-- Check for working C compiler: /usr/bin/avr-gcc
-- Check for working C compiler: /usr/bin/avr-gcc - works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Detecting C compile features
-- Detecting C compile features - done
```

(continues on next page)

(continued from previous page)

```
Using module: module_version_string
-- Configuring done
-- Generating done
-- Build files have been written to: /~/workspace/build
[root@6bf01305461f workspace]#
```

At any moment you can set configuration variables by invoking ccmake pointing to the `build` directory:

```
[root@6bf01305461f workspace]# ccmake build/
```

Modules

5.2.2 Compilation

Compilation is quite straightforward. Simply invoke `cmake` with `build` option pointing to the `build` directory:

```
[root@6bf01305461f workspace]# cmake --build ./build
Scanning dependencies of target module_version_string
[  4%] Building C object sys/kern/modules/version_string/CMakeFiles/module_version_
↪string.dir/version_string.c.obj
[  4%] Built target module_version_string
Scanning dependencies of target LazuliKernel_AVR_ATmega328p_0.1.0
[  9%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪arch/AVR/arch.c.obj
[ 13%] Building ASM object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪arch/AVR/interrupt_vectors_table.S.obj
[ 18%] Building ASM object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪arch/AVR/startup.S.obj
[ 22%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪arch/AVR/timer_counter_1.c.obj
[ 27%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪kernel.c.obj
[ 31%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪memory.c.obj
[ 36%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪scheduler.c.obj
[ 40%] Building C object sys/CMakeFiles/LazuliKernel_AVR_ATmega328p_0.1.0.dir/kern/
↪list.c.obj
[ 45%] Linking C static library libLazuliKernel_AVR_ATmega328p_0.1.0.a
[ 45%] Built target LazuliKernel_AVR_ATmega328p_0.1.0
Scanning dependencies of target LazuliUserProject_AVR_ATmega328p_1.0.0
[ 50%] Building C object CMakeFiles/LazuliUserProject_AVR_ATmega328p_1.0.0.dir/user/
↪main.c.obj
[ 54%] Linking C executable LazuliUserProject_AVR_ATmega328p_1.0.0
[ 54%] Built target LazuliUserProject_AVR_ATmega328p_1.0.0
Scanning dependencies of target hex_output
[ 59%] Generating user HEX file: LazuliUserProject_AVR_ATmega328p_1.0.0.hex
[ 59%] Built target hex_output
Scanning dependencies of target lst_ouput
[ 63%] Generating user LST file: LazuliUserProject_AVR_ATmega328p_1.0.0.lst
[ 63%] Built target lst_ouput
Scanning dependencies of target kernel_lst_ouput
[ 68%] Generating kernel LST file: LazuliKernel_AVR_ATmega328p_0.1.0.lst
[ 68%] Built target kernel_lst_ouput
Scanning dependencies of target module_clock_24
```

(continues on next page)

(continued from previous page)

```
[ 72%] Building C object sys/kern/modules/clock_24/CMakeFiles/module_clock_24.dir/
↪clock_24.c.obj
[ 72%] Built target module_clock_24
Scanning dependencies of target module_mutex
[ 77%] Building C object sys/kern/modules/mutex/CMakeFiles/module_mutex.dir/mutex.c.
↪obj
[ 81%] Building ASM object sys/kern/modules/mutex/CMakeFiles/module_mutex.dir/arch/
↪AVR/mutex.S.obj
[ 81%] Built target module_mutex
Scanning dependencies of target module_serial
[ 86%] Building C object sys/kern/modules/serial/CMakeFiles/module_serial.dir/serial.
↪c.obj
[ 90%] Building C object sys/kern/modules/serial/CMakeFiles/module_serial.dir/arch/
↪AVR/usart.c.obj
[ 90%] Built target module_serial
Scanning dependencies of target module_spinlock
[ 95%] Building C object sys/kern/modules/spinlock/CMakeFiles/module_spinlock.dir/
↪spinlock.c.obj
[100%] Building ASM object sys/kern/modules/spinlock/CMakeFiles/module_spinlock.dir/
↪arch/AVR/spinlock.S.obj
[100%] Built target module_spinlock
[root@6bf01305461f workspace]#
```

You can notice that the build system builds all modules, even those that are not used by your project. However they are built *but not linked to your project*. This is to ensure that code modifications made to the kernel doesn't break integrity of the *whole kernel*, including modules.

CHAPTER 6

Kernel documentation

You are reading the Lazuli kernel documentation.

It is highly recommended you have read all the User documentation before reading this part.

6.1 Repository tree

We give here an explanation of the main directories of the repository. It is displayed here from the root with the command `tree -d` with manual modifications to show only the most important ones. Comments on the right side have been added manually.

```
.
├── doc                         Sources of Sphinx user documentation
│   └── kernel                  Sources of Sphinx kernel documentation
├── docker                      All files related to the Lazuli Docker image
├── example-programs            Sources of examples user programs that use the Lazuli␣
↪API
├── LICENSES                    Text of the project's licenses
├── scripts                     Utility scripts
├── sys                         Base directory for all the system sources
│   ├── cmake                   CMake files, referenced by CMakeLists.txt
│   ├── include                 Base directory of user and kernel header files
│   │   └── Lazuli              Base directory of user and kernel header files
│   │       └── sys             Directory of kernel header files
│   ├── kern                    Base directory of kernel sources
│   │   ├── arch                Base directory of arch-specific kernel sources
│   │   └── modules             Base directory of kernel modules sources
│   ├── libc                    Base directory of C files related to libc implementation
│   ├── libc-headers            Base directory of standard C library header files
│   │   └── arch-dependent      Base directory of arch-dependent libc header files
│   └── unit-tests              Unit tests sources
├── templates                   File templates, used when creating new files
└── user                        Directory for the user code.
```

6.2 Memory Layout

Here is documented the memory layout for the kernel and user tasks.

In the context of AVR MCUs the term RAM used in this document refers to the MCU SRAM, and the term ROM refers to the MCU flash memory.

Memory layout is described in the file `sys/kern/linker.ld`.

6.2.1 Harvard architecture and static linking

AVR MCUs use a Harvard architecture. Machine instructions and program data are stored in separate memories.

Due to the particular architecture of AVR MCUs (especially the absence of an MMU) the kernel and user tasks are all linked in a single executable file. What we call "user" tasks are pieces of code written independently of the kernel, and that are not distributed with it. Therefore all the object files (kernel and user tasks) are statically linked and share the same memory sections.

We explain here how these sections are loaded into ROM and how the RAM is subdivided according to the sections needed at runtime.

ROM

The following sections are loaded in ROM:

- **.text**: contains all the executable code, i.e. the interrupt vectors table and the machine instructions of the user tasks and the kernel. On the AVR architecture the interrupt vectors table must be located at address 0x0000.
- **.progmem**: contains all constants that are stored in ROM. These constants can be any type of data (strings, structs, ...). In C code, a constant is marked to be stored in ROM using the `PROGMEM` macro. When a constant is needed during execution, it is loaded in RAM at runtime using special AVR `lpm` instruction.
- **.data**: contains all global variables that are initialized in C. The .data section doesn't contains machine code but program data so it needs to be loaded in RAM at startup (see Startup.md).
- **.rodata**: contains all global variables that are initialized in C and marked as readonly using the keyword `const`. The .rodata section doesn't contain machine code but program data so it needs to be loaded in RAM at startup (see Startup.md).

This graph shows how these sections are loaded into ROM. The arrows on the right specify addresses.

```
+----------------+
|. . . . . . . .|<- SIZEOF(ROM) - 1
| . . . . . . . .|
|. (unused). . . |
| . . . . . . . .|
|. . . . . . . . |
| . . . . . . . .|<- _rodata_load_start + SIZEOF(.rodata)
+----------------+
|                |<- _rodata_load_start + SIZEOF(.rodata) - 1
|  .rodata       |
|  section       |   _data_load_start + SIZEOF(.data)
|                |<- (_rodata_load_start in the linker)
+----------------+
|                |<- _data_load_start + SIZEOF(.data) - 1
|  .data         |
|  section       |   _progmem_load_start + SIZEOF(.progmem)
```

(continues on next page)

(continued from previous page)

```
|                |<- (_data_load_start) in the linker)
+----------------+
|                |<- _progmem_load_start + SIZEOF(.progmem) - 1
|  .progmem      |
|  section       |
|                |<- SIZEOF(.text) (_progmem_load_start in the linker)
+----------------+
|                |<- SIZEOF(.text) - 1
|                |
|  .text         |
|  section       |
|                |
|                |
+-  -  -  -  -  -+
| interrupt      |
| vectors        |
| table          |<- 0x0
+----------------+
```

RAM

A small part of the RAM is mapped to machine registers and I/O registers. The usable memory part starts at address 0x100 and contains the following regions:

- **.data**: This section is fixed in size and is copied from ROM during startup. It contains all initialized global variables.
- **.rodata**: This section is fixed in size and is copied from ROM during startup. It contains all global constants.
- **.bss**: This section is fixed in size and contains all uninitialized global variables in C. It is set to zero during startup.
- **.noinit**: This section contains all variables marked with the macro `NOINIT`. This macro is used to indicated that an uninitialized global variable must not be initialized to zero at startup.
- The **heap**: This is the kernel heap. After kernel startup the heap size is 0.
- The **kernel stack**: During system startup only the kernel stack exists. On the AVR architecture the stack grows downward. The stack pointer points to the next free memory location that will be used when performing a `push`.
- The space between the heap and the stack is left unused. This space isn't fixed in size because both the heap and the stack can grow downward or upward at runtime.

This graph shows how these sections are loaded into RAM: The arrows on the right specify addresses.

```
+----------------+
|                |<- SIZEOF(RAM) - 1 (_ramend in the linker)
| stack          |
|                |
+----------------+
|. . . . . . . .|<- stack pointer (known at runtime)
| (unused). . . .|
|. . . . . . . .|
| . . . . . . . .|<- break (known at runtime)
+----------------+
|                |
```

(continues on next page)

(continued from previous page)

```
| heap            |
|                 |<- heap_start (_brk in the linker)
+-----------------+
| .noinit section|
| (non zero'd at  |
| startup)        |
|                 |<- 0x100 + SIZEOF(.data) + SIZEOF(.rodata) + SIZEOF(.bss)
+-----------------+
| .bss section    |<- 0x100 + SIZEOF(.data) + SIZEOF(.bss) - 1
| (zero'd at      |
| startup)        |   0x100 + SIZEOF(.data) + SIZEOF(.rodata)
|                 |<- (_bss_start in the linker)
+-----------------+
| .rodata section|<- 0x100 + SIZEOF(.data) + SIZEOF(.rodata) - 1
| (loaded from    |
| ROM)            |<- 0x100 + SIZEOF(.data) (_rodata_start in the linker)
+-----------------+
| .data section   |<- 0x100 + SIZEOF(.data) - 1
| (loaded from    |
| ROM)            |<- 0x100 (_data_start in the linker)
+-----------------+
| machine and     |<- 0xff
| I/O registers   |
|                 |<- 0x0
+-----------------+
```

6.2.2 Allocating user tasks

The allocation of new user tasks is done in the function Lz_Scheduler_RegisterTask(). When registering a new user task the kernel must allocate enough space to contain the *Task* object that represents a task. The kernel then allocates enough space for the task's stack. The reserved space for a task's stack is fixed in size, so a user task cannot grow its stack bigger than the size asked when registering.

6.3 System startup

6.3.1 Assembly - Setting up a C runtime

System startup is divided in 2 parts:

- The first part, written in assembly language, executes right after powering on the system, and has the responsibility to set up an operational C runtime.
- The second part, mainly written in C, performs various operations that are necessary to initialize the data structures of the kernel, before giving hand to user code.

Initial system startup is written is assembly language, in the file `sys/kern/arch/AVR/startup.S`. It contains the necessary code to launch the kernel. The main goal of system startup routines is to set up a working C runtime environment.

When powered on the AVR MCU starts executing code at address 0x0000 in ROM. This entry point is defined as the reset interrupt handler (written in `sys/kern/arch/AVR/interrupt_vectors_table.S`). This interrupt handler performs a jump to the ASM routine `reset_system`. This routine makes heavy use of symbols defines by the linker script (in the file `sys/kern/linker.ld`).

First of all, the routine `reset_system` will make sure that global interrupts are disabled, then set the stack pointer to the end of RAM. This stack that will be used during all the initialization process. We will then set up the appropriate sections in RAM, in the following order:

- `bss`: Set all the section to zero.
- `data`: Copy data from ROM.
- `rodata`: Copy data from ROM.

After these operations, we now have an operational C runtime. That's all for the assembly part. The `reset_system` routine will finally give hand to C code, by calling the C function `Kernel_Main` defined in `sys/kern/kernel.c`.

6.3.2 C - Initializing kernel structures

We are now in `Kernel_Main`.

6.4 Instrumentation and diagnostics

Hi there!

6.4.1 Stack usage

Hello, stack usage!

6.4.2 Context switches instrumentation

Hello, context switches instrumentation!

6.5 Version management

6.6 Contributing

Thanks for contributing to Lazuli!

Contributions are welcome!

However, contributions must follow a set of rules.

It doesn't matter how big your contribution is: it can be fixing a bug, fixing a spelling mistake or a typo, suggesting new features, or implementing new functionalities to the kernel. Any kind of contribution is welcome!

6.6.1 Contribute without coding

You can use the `Issues` tab (https://github.com/randruc/Lazuli/issues) of the project to report troubleshooting, spelling mistakes or typos, ask questions, suggest new features, or anything you have in mind!

6.6.2 Contribute to the code

The official code repository for the project is on GitHub at https://github.com/randruc/Lazuli. GitHub is used to manage all the project: code, pull requests, issues, etc.

Before contributing, make sure you've read :

- The page *About documentation* if you wish to contribute to the documentation.
- The page *Code style and conventions* if you wish to contribute to the code.

When coding, make sure you've updated all the appropriate files and lines of text apart from the code itself. That includes code comments, documentation files, CMakeLists.txt, or any other file that could be impacted by your change.

Contributions is done using the classic GitHub pull-request mechanism:

- Fork the repository.
- Create a branch *with a meaningful name* that will bear your contribution.
- Commit your contribution on your branch. Be careful to respect to expected format for commit messages. Read below for the detail.
- Push your branch, then do the pull request. If your contribution is not obvious, it is recommended to explain it with a message.
- Your contribution will then be reviewed. Be aware that questions can be asked or remarks can be made before the ultimate merging.

For now, squashing and rebasing are disabled for pull-requests.

Commit messages

The expected format for commit messages is the following:

```
<Part or module>: <Brief explanation on one line.>

<Detailed explanation if needed.>
<The detailed explanation can be expressed as a list.>
```

Here are a few examples of real commit messages that follow this format:

(from commit 5cca0fd69947409221b6337c98c6b1e3ac9419ef)

```
GitHub Actions: Set the "pipefail" bash option when building docker image.

This is needed because we pipe the output of "docker build" to "ts". So if
"docker build" fails, the exit code is not the one of "ts".
```

(from commit d31a215cf821b84804cbadf5169711d6c4ccc6a9)

```
stack_usage.sh: Improvements on the usability.

* Command line options have been added to let the user sort by stack size or
  by file name.
* The output is no longer piped into less.
```

Continuous Integration

Continuous Integration (CI) is configured on the official GitHub repository using GitHub Actions, and is triggered on pull-requests.

The project's CI performs the following tasks:

- Execute `scripts/checklines.sh` to check for trailing whitespaces, line length, etc.
- Build the Lazuli development environment Docker image from the root `Dockerfile`.
- In the newly built Docker image, configure then build Lazuli in its default configuration (reminder: modules are always built even if not selected to be linked to the final binary).
- Build Doxygen documentation.
- Build Sphinx documentation.

All the CI pipeline is configured to treat warnings are errors. A pull-request with a failing CI has no chance to be merged.

6.7 Code style and conventions

6.7.1 C style and conventions

We explain here the programming style and conventions to apply on C code for the project.

As a general rule, the code must be clear and readable. Reading it must be a pleasant experience. The code must not contain useless or redundant things.

C dialect

The whole project is written in pure ANSI C 89. All C code files are encoded in raw 7-bit ASCII.

Note: This is to the allow C code to be ported easily to many platforms and architectures, by being compiled by the largest number of compilers. Another reason is that C89 is understood by all C code-checking tools.

Whitespaces, tabs and newlines

Never use tabs to indent C code. C code must use 2 spaces for indentation.

Note: Every editor must display the code the same way.

Lines mustn't have trailing whitespaces.

Files must end with a newline character.

Note: The code file must print correctly when using tools like `cat` or `less` in the console.

Line length

Line length is set to 80 characters. Each line of code shouldn't exceed 80 columns.

Note: Documents must be easily readable without horizontal scrolling in a console (e.g. with `less` or `cat`), in simple text editors, or when comparing two versions in a file diff utility.

Comments

Only use C89 comments, no C99/C++ comments.

Note: This is to be compliant with C89.

Never write nested comments. Never write "pretty typos". Never comment out code unless you want to document something.

Use only one of the 2 following forms. 1-line comments can use both allowed forms, while multiple line comments must use the second form only.

This is good:

```
/* First form. This is a comment */

/*
 * Second form.
 *
 * This is a comment
 * that spans over
 * multiple lines.
 */

/*
 * The second form can be used for 1-line comments.
 */
```

This is *not* good:

```
// This is a comment.

/* This is a comment
   that spans over
   multiple lines */

 /*************************************************
  *             Wonderful pretty typo              *
  *************************************************/
```

Where to put comments

Good code shouldn't need comments. Good code could should express by itself.

As a general rule:

- Always comment the API of every function, even static ones
- Avoid comments inside functions

Use comments inside functions very carefully only to explain something that is not obvious for the reader.

Every function must be documented using Doxygen tags. For static functions, this is done above the function itself. For public functions, this is done above the function prototype declaration in the header file only.

Return types

The return type of a function must be declared on the line preceding the function name.

This is good :

```
int
sum(int a, int b)
{
  return a + b;
}
```

This is *not* good :

```
int sum(int a, int b)
{
  return a + b;
}
```

Note: C code must be easily readable in console or with editors that don't support syntax highlighting.

Curly braces

Curly braces use the K&R style.

For functions, opening and closing braces are always on their own line.

This is the only style allowed for functions:

```
int SayHello(void)
{
  printf("Hello world\n");

  return 0;
}
```

For control statements the opening braces are always on the same line than the statement and the closing braces are always on a new line.

This is the only style allowed for control statements:

```
if (n == 4) {
  printf("Bye");

  return 0;
}
```

This applies to all control statements that involve code sections: `if`, `else`, `for`, `while`, `do`.

For if-else blocks, the else keyword must be on the same line than the braces.

This is the only style allowed for if-else blocks:

```
if (n == 4) {
  j += 8;
  printf("Bye");
} else {
  j = 0;
  i++;
}
```

For control statements that involve code section, braces must *always* be present, even for one-line expressions:

```
if (NULL == map) {
  return NULL;
}

for (i = 0; i < LENGTH; i++) {
  t[i] = NULL;
}

if (NULL == map) {
  return NULL;
} else {
  return map->next;
}
```

Note: Always putting the braces helps to avoid some bugs. As an example, you can read this article about the famous case of Apple's SSL/TLS bug: https://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/

6.8 Best practices

Here are listed individual pages about best practices to consider when writing C code in Lazuli.

6.8.1 Think twice before using enumerations in C

It is generally considered a good practice to define named integer constants as enumerations (`enum`) if they are related to each other. i.e.: they can be grouped under a common name, bringing more meaning, more "semantic intention":

```
enum Planet {
  MERCURY = 0xa1,
  VENUS = 0xb2,
  EARTH = 0xc3,
  MARS = 0xd4
};
```

as opposed to preprocessor's `#define` that cannot group a set of macro constants under a common name:

```
#define MERCURY (0xa1)
#define VENUS (0xb2)
#define EARTH (0xc3)
#define MARS (0xd4)
```

Most of the time and for the vast majority of usual C code, this practice should be applied. Enumerations also have the advantage of being true C symbols from the compiler and debugger point of view.

However, there are a few situations where manipulating integer constants as enumerations should not be considered, or at least should be taken really carefully. These situations happen when enumerations are manipulated as l-values on specific architectures.

The C specification

This is an excerpt from the C specification about enumerations (in http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf - 6.7.2.2 Enumeration specifiers):

> *"The expression that defines the value of an enumeration constant shall be an integer constant expression that has a value representable as an int."*

As every C programmer knows, the size of an `int` in C is left unspecified by the standard. This means that the size of a enumeration constant value is platform/architecture/compiler dependant. Enumerations are not of a guaranteed size by the C specification, and there is no standard way to do so.

A consequence on binary interfacing

The first consequence of the C specification is that `enum` constants don't have a fixed size across different architectures, as the size of an `int` varies between architectures/compilers. This is quite a well-know fact amongst C programmers.

Defining constants as `enum` should be avoided when binary interaction/interfacing is needed accross different architectures (e.g. over networks).

An argument against *specific* uses of enumerations

Let's now introduce a sneaky pitfall, that happens *under specific conditions*. Lazuli RTOS is targeting embedded systems. It is written in ANSI C and aims to be easily portable across different architectures.

We take here the example of the AVR architecture, which is a target for the Lazuli RTOS.

AVR is an 8-bit CPU, but on which many compilers (such as AVR-GCC) define the type `int` to be 16-bit long. This means that enumerations will be 16-bit long *as well*.

Here is our general rule:

> **Great care must be taken when using enumerations on architectures where the size of the machine word is narrower than the size of an int.**

Although this situation is usually not a problem (we've used 64-bit variables on 32-bit machines for years), it can become one in certain contexts, and can even lead to synchronization problems as we will see. And it's a situation that happens quite often in embedded C development, as we are often dealing with tiny machines. It is much more difficult to see these problems and their consequences when using enumerations rather than integers, because we often rely on standard header `<stdint.h>` when declaring integer variables. We then have a convenient way to master the size of integer variables right from their declaration. Unfortunately no equivalent exists for enumerations, and its not easy to spot their size at a glance when reviewing the source code.

Using enumeration variables on a platform where the size of the machine word is narrower than the size of an int has 3 main consequences, that are in fact linked together:

- **An impact on performance**
- **An impact on memory usage and the total size of the resulting binary**
- **Non atomicity of memory accesses**

Now let's see why.

We consider the following piece of C code (the constant values have been chosen so we can find them easily within the disassembly):

```
enum Planet {
  MERCURY = 0xa1,
  VENUS = 0xb2,
  EARTH = 0xc3,
  MARS = 0xd4
};

volatile enum Planet planet;

void
example(void)
{
  planet = EARTH;
}
```

Notice that all these constants are only 8-bit long.

Let's compile this short example for the AVR architecture and observe the assembly generated by the compiler.

```
 [root@0ca45bbdd5b1 best_practices]# avr-gcc -c -O3 enum_example.c \
 > && avr-objdump -d enum_example.o

 enum_example.o:     file format elf32-avr


 Disassembly of section .text:

 00000000 <example>:
    0:   83 ec           ldi r24, 0xC3   ; 195
    2:   90 e0           ldi r25, 0x00   ; 0
    4:   90 93 00 00     sts 0x0000, r25 ; 0x800000 <__SREG__+0x7fffc1>
    8:   80 93 00 00     sts 0x0000, r24 ; 0x800000 <__SREG__+0x7fffc1>
    c:   08 95           ret
```

First, the value of the constant `EARTH` (`0xc3`) is loaded in 2 registers `r24` and `r25` (`ldi`, LoaD-Immediate). Notice here that one register (`r25`) is loaded with zero, as it is the high byte of `0xc3` encoded as a 16-bit word. Then this value is stored in memory, the corresponding lines are highlighted. You can also notice that, even with compiler optimizations switched on (`-O3`), the actual writing in memory is done in 2 operations (`sts`, Store-To-Sram), one byte at a time. This is coherent, as we are storing a 16-bit value in memory using an 8-bit CPU.

Now let's do the same thing, but by replacing the `enum` for a smaller integer type that fits the architecture word size, and using the C preprocessor to define our named constants:

```
typedef unsigned char planet_t;

#define MERCURY ((planet_t)0xa1)
#define VENUS ((planet_t)0xb2)
#define EARTH ((planet_t)0xc3)
#define MARS ((planet_t)0xd4)
```

(continues on next page)

(continued from previous page)

```
volatile planet_t planet;

void
example(void)
{
  planet = EARTH;
}
```

Let's examine the disassembly:

```
 [root@0ca45bbdd5b1 best_practices]# avr-gcc -c -O3 enum_example_with_cpp.c \
 > && avr-objdump -d enum_example_with_cpp.o

 enum_example_with_cpp.o:     file format elf32-avr


 Disassembly of section .text:

 00000000 <example>:
    0:   83 ec           ldi r24, 0xC3   ; 195
    2:   80 93 00 00     sts 0x0000, r24 ; 0x800000 <__SREG__+0x7fffc1>
    6:   08 95           ret
```

The actual writing in memory is now performed in one operation. Needless to precise that this writing operation is atomic on the AVR.

We can now address our three concerns from before:

First, about **performance**. This is a strong concern if you are writing code that must be *fast* in those very architecture-specific cases (which is the case of many embedded projects, such as Lazuli RTOS, or cross-architecture projects). If the values of your `enum` can all fit within a machine word, then you should consider using `#define` with a machine word size instead of `enum`. In the Lazuli kernel, all enumerations definitions used in the scheduler have been replaced by `typedef` and `#define` to a smaller type.

Then, about the **memory usage and total size of binary**. It is obvious when comparing the two disassemblies above that using enumerations where a smaller type can be used leads to a bigger memory usage. This is true for ROM (program memory), as well as RAM usage (global variables, stack variables, etc.). The difference may only be one byte, but the AVR ATmega328P only has 2048 bytes of RAM, so one byte *is* important. Obviously, this can be applied only if the values of your `enum` can all fit within a machine word.

And last but not least, about the **atomicity of memory accesses**. Looking at the first disassembly above, a context switch or an interrupt handler can occur during writing in memory, between the two `sts`. If another thread of execution tries to access the value while the writing thread is suspended between the two `sts` of its writing operation, then the reader thread will possibly read a corrupted value. This can have catastrophic consequences. This means that *in cases where the size of the machine word is narrower than the size of an int*, enumerations *must not* be used to pass signals or messages between different threads of execution. Otherwise synchronization issues can occur.

Replacing enumerations by `typedef` and `#define` to a type whose size fits the machine word solves this problem.

In those cases, you can use the type `u_read_write_atomic_t` that is defined in `<Lazuli/common.h>`. The type `u_read_write_atomic_t` is an unsigned integer having a size that guarantees that reading/writing from/to memory is atomic accross all architectures. This is the equivalent of the C standard `sig_atomic_t` defined in `<signal.h>`. In some situations, using this type can also avoid using an explicit lock.

This type is used in the Lazuli kernel any time a user task needs to pass a message to the scheduler/kernel.

A note on `-fshort-enums`

Some compilers support an option to produce "short enumerations", such as GCC's `-fshort-enums`. This option is not used in Lazuli, as it compiler-specific. Lazuli is written in standard C89, so it must compile and run correctly with any compiler. *No assumptions about the compiler, no surprises.* The code **is not** supposed to work correctly only under the condition of being compiled with a specific compiler (especially for atomic reads and writes).

Some other problems can occur when using `-fshort-enum`, but it is not worth speaking about them here, as we care about not being compiler-specific. You can read more about the pitfalls of using `-fshort-enums` here: https://interrupt.memfault.com/blog/best-and-worst-gcc-clang-compiler-flags#-fshort-enum

6.9 About documentation

The documentation for the projects comes from 2 different sources:

- **Textual documentation**, generated with **Sphinx** from text files written in rst format.
- **API documentation**, generated with **Doxygen** from formatted code comments.

The documentation is rendered in 2 formats:

- **HTML**
- **man**: The man page output is used in the Lazuli Docker image. Any developer using the development environment Docker image can type `man Lazuli` to read the exact complete documentation you are now reading. Thanks to Doxygen, the user can also use `man` to obtain the documentation about any function or symbol used in the project.

6.9.1 Sphinx and reStructuredText

The directory `doc/` in the repository source tree contains all the official documentation of the project. This is the one you are now reading.

The Lazuli project aims to be well-documented. Documentation must be up-to-date and easy to find.

This documentation is written in reStructuredText (rst) and built with Sphinx. The official website of sphinx is https://www.sphinx-doc.org

reStructuredText file markup allows to write documents that can be read in text mode within a console as well as in a modern browser, while benefiting of a nice formatting on the two supports.

Building Sphinx documentation

To build Sphinx documentation, simply go to the `doc/` directory, then type `make`. This will build the full documentation in HTML and man formats. The output will be in the directory `doc/_build`.

To build the documentation only in a specific format, simply type `make html` or `make man`.

Style and conventions used in documentation files

Files must be named using lower camel case with underscores as separators and have the extension `.rst` (e.g. the file your are reading is called `about_documentation.rst`).

Except in some very specific situations (e.g. when writing URLs), line length is set to 80 characters. Documents must be easily readable without horizontal scrolling in a console (e.g. with `less`), in simple text editors, in a file diff utility,

or on any platform that don't natively supports rst syntax. 80 columns line breaks are not a problem when rendering to HTML or man pages as rst ignores line breaks when processing.

Always keep in mind that the file can be read in one of these two possible ways, so you should regularly check how it looks on both supports when writing.

Headings are always declared with underlining. Heading characters are used in the following order:

1. =
2. -
3. ^

There must be only one heading of level 1 per page.

6.9.2 Doxygen

The other kind of documentation comes from code comments with Doxygen commands. The official website of Doxygen is https://www.doxygen.nl

The API documentation for the project is hosted at https://randruc.github.io/Lazuli/doxygen/latest/

Code comments

All C code of the project is documented using Doxygen code comments. The absence of Doxygen code comments will cause an error during build.

Building Doxygen documentation

To build Doxygen documentation, simply go to the root of the repository, then type `doxygen`. This will build documentation in HTML and man formats. The output will be in the directory `doxygen_output`.

CHAPTER 7

Glossary

Entry 1

This is the definition of entry 1.
It can span over multiple lines.

Entry 2 This is the definition of entry 2. It can span over multiple lines.

If you wish to read the API documentation, please browse to https://randruc.github.io/Lazuli/doxygen/latest/

Index

E